Modo de direccionamiento base y Variables locales en la pila

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

Modo de direccionamiento base

En este modo de direccionamiento el cálculo de la dirección efectiva del dato emplea uno de los registros **BX** o **BP**. Las referencias a **BX** son desplazamientos con respecto al registro de segmento de datos **DS**, mientras que las referencias a **BP** son desplazamientos con respecto al registro de segmento de pila **SS.**

El modo de direccionamiento base tiene la siguiente sintaxis:

[bx+*n*]
[bx-*n*]
[bp+*n*]
[bp-*n*]

En los dos primeros casos el desplazamiento del dato con respecto a **DS** está dado por el valor de **BX más o menos *n bytes*.**
*En los dos últimos casos el desplazamiento del dato con respecto a **SS*** está dado por el valor de **BP más o menos *n bytes*.**

El direccionamiento base utilizando el registro **BX** se emplea normalmente para localizar campos dentro de una estructura de datos.

Por ejemplo:

```
mov     bx, offset dato ; Apunta a la estructura
                        ; dato
mov     ax, [bx+5]      ; Obtén el valor que se
                        ; encuentra a 5 bytes
                        ; del inicio de dato
inc     [word bx+3]     ; Incrementa el valor de
                        ; tipo palabra que se
                        ; encuentra a 3 bytes del
                        ; inicio de dato
```

Nota.- El direccionamiento base utilizando el registro **BP** se emplea normalmente para referenciar a variable locales en la pila del programa como se verá posteriormente

Variables locales en la pila

Los procedimientos en ensamblador utilizan por lo general los registros de propósito general para variables locales. En la mayoría de los casos los registros con que disponemos: **AX, BX, CX, DX, SI, DI,** son suficientes para nuestras necesidades de almacenamiento temporal.

Sin embargo si llegáramos a necesitar de más espacio de almacenamiento temporal podemos emplear el mecanismo usado por los lenguajes de alto nivel para ubicar las variables locales de un procedimiento.

Los lenguajes de alto nivel crean a las variables locales de un procedimiento en la pila del programa.

El mecanismo empleado para crear las variables en la pila es el siguiente:

1) En la figura 4, se muestra el estado de la pila inmediatamente después de entrar al procedimiento. Note que en la localidad apuntada por el tope de la pila se encuentra la dirección de regreso del procedimiento *dirRegProc*.

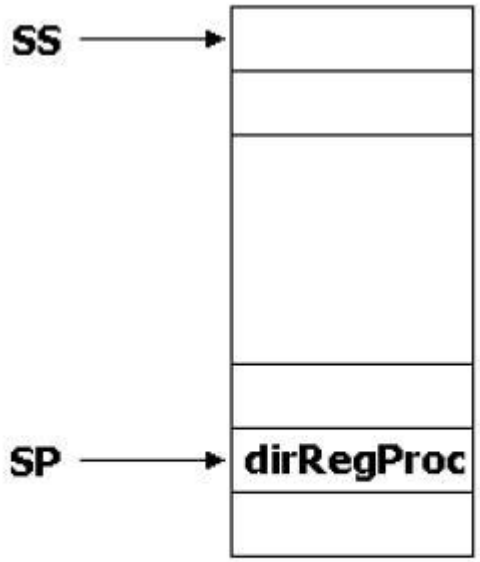
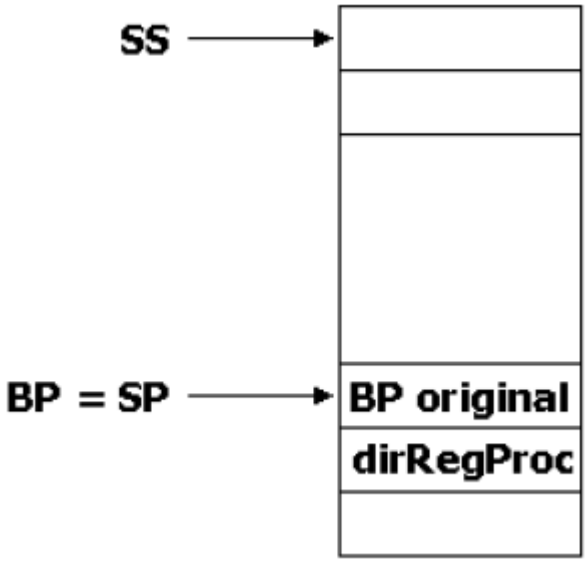


Figura 4

2) Una vez dentro del procedimiento, lo primero que se hace es preservar el valor del **Registro Apuntador de Base**, **BP**, insertándolo en la pila. A continuación se hace que **BP** apunte al tope de la pila, ya que a partir de esa posición se ubicarán las variables locales. Las instrucciones para obtener lo anterior son:

```
push    bp
mov     bp, sp
```

El estado de la pila, después de estas operaciones se muestra en la figura 5:

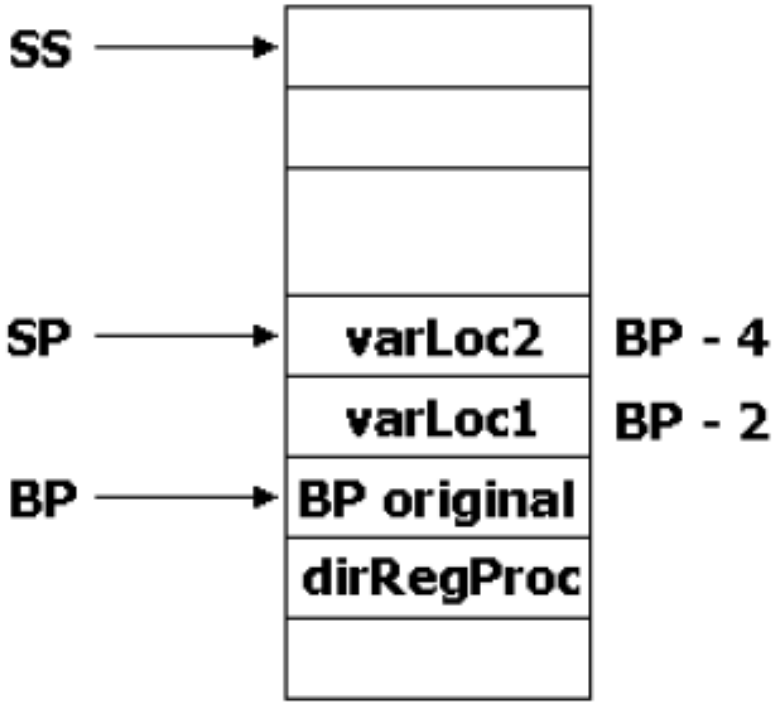


3) Se crean las variables locales decrementando **SP** en el número de bytes requeridos por las variables. Por ejemplo, para crear dos variables locales de tipo palabra, la instrucción sería:

```
sub sp, 4                    ; 2 variables de 2 bytes
                             ; c/u = 4 bytes
```

El estado de la pila después de la instrucción se muestra en la figura 6. Las variables locales se identifican en la figura con los nombres: **varLoc1 y varLoc2**. Sin embargo el acceso a las variables es mediante su dirección las cuales son: **BP-2** y **BP-4**, respectivamente.

Figura 6.



El mecanismo empleado para destruir las variables es el siguiente:

1) Se incrementa **SP** en el número de bytes requeridos por las variables locales. Esto se puede hacer mediante la instrucción:

```
mov     sp, bp          ; Equivale en este caso
                        ; a: add sp, 4
```

Esto nos deja la pila como en la figura 5.

2) Se recupera el valor de BP de la pila, haciendo que **SP** apunte de nuevo a la dirección de regreso del procedimiento, mediante la instrucción:

```
pop     bp
```

El estado final de la pila se muestra en la figura 4.

Ejemplo sobre variables locales en la pila:

El siguiente procedimiento reduce una fracción a su expresión más simple. Por ejemplo el número 12/60 se reduce a 1/5, dividiendo el numerador y denominador entre el máximo común divisor.

```
;*****************************************************
; REDFRAC
;
; Este procedimiento reduce una fracción a su mínima
; expresión
;
; Parámetros:
;
;       AX = numerador
;       DX = denominador
;
; Regresa:
;
;       AX = numerador reducido
;       DX = denominador reducido
;
; El pseudocódigo para este procedimiento es:
```

```
; El pseudocódigo para este procedimiento es:

;    BX = mcd(numerador, denominador)
;
;    numerador /= BX
;    denominador /= BX
;
;*****************************************************************
proc    redfrac
        push    bp                  ; Preserva BP
        mov     bp, sp              ; Crea variables
        sub     sp, 4               ; locales en la pila
        push    bx                  ; Preserva BX

;num = [BP-2], den = [BP-4]
        mov     [bp-2], ax          ; num = numerador
        mov     [bp-4], dx          ; den = denominador

        call    mcd                 ; BX = mcd(numerador,
                                    ;           denominador)
        mov     bx, ax
        xor     dx, dx              ; num = numerador/BX
        mov     ax, [bp-2]
        div     bx
        mov     [bp-2], ax
```

```
        mov     ax, [bp-4]      ; DX = denominador/BX
        div     bx
        mov     dx, ax
        mov     ax, [bp-2]      ; AX = num

        pop     bx              ; Restaura BX
        mov     sp, bp          ; Elimina variables locales
        pop     bp              ; Restaura BP
        ret
endp    redfrac
```

Variables locales en la pila usando la directiva local:

El ensamblador permite simplificar un poco el manejo de las variables locales permitiendo emplear identificadores locales para referenciar a las variables en lugar de emplear el direccionamiento base. Para ello se emplea la directiva **local.** La directiva local le asigna a cada variable local creada en la pila del programa un identificador local.
La sintaxis de esta directiva es:

```
local nomVar: tipo[, nomVar: tipo]... [ = tamVarsLoc]
```

donde ***nomVar*** *es el nombre de la variable y tipo es su tipo:* ***byte, word, dword, qword,*** *etc.*

tamVarsLoc *es un símbolo que toma el valor del número de bytes ocupado por las variables locales.*
Una vez que se ha empleado la directiva **local s**e puede hacer referencia a las variables locales mediante sus nombres como si fueran variables globales.

Ejemplo sobre variables locales en la pila usando la directiva local

El siguiente procedimiento es una modificación del ejemplo anterior en el que se utiliza la directiva **local** para darle nombres locales a las variables locales creadas en la pila.

```
;*************************************************************
; REDFRAC2
;
; Este procedimiento reduce una fracción a su mínima
; expresión
;
; Parámetros:
;
;       AX = numerador
;       DX = denominador
;
; Regresa:
;
;       AX = numerador reducido
;       DX = denominador reducido
;
; El pseudocódigo para este procedimiento es:
;
;   numerador /= BX
;   denominador /= BX
;*************************************************************
```

```
proc    redfrac
; Declara las variables locales en la pila
        local   num: word, den: word = tamVarsLoc

        push    bp                  ; Preserva BP
        mov     bp, sp              ; Crea variables
        sub     sp, tamVarsLoc      ; locales en la pila
        push    bx                  ; Preserva BX

        mov     [num], ax           ; num = numerador
        mov     [den], dx           ; den = denominador

        call    mcd                 ; BX = mcd(numerador,
        mov     bx, ax              ;           denominador)

        xor     dx, dx              ; num = numerador/BX
        mov     ax, [num]
        div     bx
        mov     [num], ax

        mov     ax, [den]           ; DX = denominador/BX
        div     bx
```

```
        mov     dx, ax
        mov     ax, [num]       ; AX = num


        pop     bx              ; Restaura BX
        mov     sp, bp          ; Elimina variables
                                ; locales

        pop     bp              ; Restaura BP
        ret
endp    redfrac
```

Programación Modular

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

En los programas en ensamblador hechos hasta ahora, todo el código del programa reside en un sólo archivo. Sin embargo es posible y en muchos casos deseable dividir en código de un programa en varios archivos llamados módulos. Hay varias ventajas de emplear esta técnica llamada programación modular:

✓Hay un límite en el tamaño de un archivo que un programa ensamblador puede manejar. Si el código del programa crece mucho, al dividirlo en módulos, tendremos archivos de menor tamaño. Como cada módulo puede ensamblarse por separado, esta técnica nos permite ensamblar programas más grandes.

✓ El proceso de ensamblado de un módulo es menor que el de un programa completo. Cuando se está en la etapa de construcción / depuración de un programa, por lo general uno escribe y depura un procedimiento a la vez, por lo que sólo se está haciendo cambios a un módulo, por lo que sólo ese módulo requiere de volver a ensamblarlo.

✓Para obtener un programa ejecutable, los códigos objeto obtenidos al ensamblar cada módulo deben ligarse. Normalmente, cada módulo contiene un conjunto de procedimientos relacionados. Estos procedimientos pueden emplearse para formar diferentes programas, simplemente ligándolos con otros módulos.

✓Podemos utilizar otros módulos escritos por terceros para resolver problemas específicos.

✓ Podemos escribir programas escritos en lenguaje de alto nivel y que utilicen módulos escritos
en ensamblador.

Cada módulo en ensamblador puede contener declaraciones de constantes, declaraciones de variables y definiciones de procedimientos. La estructura de cada módulo es similar a la estructura de un programa sin módulos con las siguientes diferencias:

- Sólo uno de los módulos contiene las instrucciones que inicializan el segmento de datos y las instrucciones que terminan el programa y regresan el control al sistema operativo. A este módulo se le conoce como el módulo principal.

- Todos los módulos tienen código de inicio con las directivas: **ideal, dosseg, model,** pero sólo el módulo principal tiene la directiva **stack.**

- Todos los módulos tiene la directiva **end** que indica el final del código de ese módulo. Sin embargo sólo en el módulo principal la directiva **end** va seguida de la etiqueta que apunta al punto de entrada del programa. En los otros módulos la directiva **end** aparece por sí sola.

➢Las constantes, variables y procedimientos declaradas y definidos en un módulo pueden ser conocidas sólo en el módulo donde fueron declaradas o pueden ser conocidas en otros módulos también. Los símbolos de las constantes, variables y procedimientos de un módulo que deseamos que se conozcan en otros módulos deben de exportarse utilizando la directiva **public.**

La sintaxis de la directiva public es:

```
public nomSimb[, nomSimb] ...
```

donde *nomSimb es el nombre de la constante, variable o procedimiento cuyo símbolo se va* exportar. Sólo los símbolos de las constantes numéricas declaradas con la directiva = pueden exportarse.
Los símbolos de las constantes declaradas con la directiva **equ** no pueden exportarse.

➢Los símbolos de las constantes, variables y procedimientos declaradas y definidos en otro módulo y que fueron exportados con la directiva **public** tienen que importarse en el módulo en que se desea que sean conocidos también. Para importar un símbolo se utiliza la directiva **extrn**, cuya sintaxis es:

```
extrn nomSimb: tipo[, nomSimb: tipo] ...
```

donde *nomSimb es el nombre de la constante, variable o procedimiento cuyo símbolo se va* importar.

tipo especifica el tipo de símbolo. Para los símbolos de las constantes numéricas declaradas con la directiva = el tipo es **abs**; para los símbolos de las variables a importar los tipos son: **byte, word, dword, qword,** etc. y para los procedimientos el tipo es **proc.**

Ejemplo que muestra parte del código de un programa formado por dos módulos: MODULO.ASM y PRINCIPA.ASM

```
;*************************************************************
; MÓDULO.ASM
;
; Este módulo exporta algunos de sus símbolos e importa
; algunos símbolos del módulo con el programa principal.
;*************************************************************

;****** CÓDIGO DE INICIO ***********************************

        ideal
        dosseg
        model   small

;****** DECLARACIÓN DE CONSTANTES SIMBÓLICAS ***************

        extrn   cte4: abs       ; importa cte4
        public  cte2            ; exporta cte2

cte1    equ     10              ; constante local
cte2    =       20              ; constante global
```

```
;****** VARIABLES DEL MODULO ********************************

        dataseg

        extrn   dato4: byte     ; importa dato4
        public  dato2           ; exporta dato2

dato1   db      ?               ; variable local
dato2   dw      ?               ; variable global

;****** CÓDIGO DEL MÓDULO ************************************

        codeseg

        extrn   proc4: proc     ; importa proc4
        public  proc2           ; exporta proc2

;****** PROCEDIMIENTOS ***************************************

;*************************************************************
; PROC1
;
; Este procedimiento es local a este módulo.
;*************************************************************
proc    proc1
        ...
        ret
endp    proc1
```

Procedimiento 2 (Global)

```
;*************************************************************
; PROC2
;
; Este procedimiento es global.
;*************************************************************
proc    proc2
        ...
        ret
endp    proc2

;****** CÓDIGO DE TERMINACIÓN ***************************

        end
```

Módulo con el Programa Principal

```
;*************************************************************
; PRINCIPA.ASM
;
; Este módulo que contiene el programa principal, también
; exporta algunos de sus símbolos e importa algunos
; símbolos del otro módulo.
;*************************************************************

;****** CÓDIGO DE INICIO ***********************************

        ideal
        dosseg
        model   small
        stack   256

;****** DECLARACIÓN DE CONSTANTES SIMBÓLICAS ***************

        extrn   cte2: abs       ; importa cte4
        public  cte4            ; exporta cte2

cte3    equ     10              ; constante local
cte4    =       20              ; constante global
```

```
;****** VARIABLES DEL MÓDULO ******************************

        dataseg

        extrn   dato2: word     ; importa dato4
        public  dato4           ; exporta dato2

dato3   db      ?               ; variable local
dato4   db      ?               ; variable global

;****** CÓDIGO DEL MÓDULO **********************************

        codeseg

        extrn   proc2: proc     ; importa proc2

        public  proc4           ; exporta proc4

inicio:
        mov     ax, @data       ; Inicializa el segmento
        mov     ds, ax          ; de datos
        ...

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h
```

```
;****** PROCEDIMIENTOS ************************************

;*****************************************************************
; PROC3
;
; Este procedimiento es local a este módulo.
;*****************************************************************
proc    proc3
        ...
        ret
endp    proc3

;*****************************************************************
; PROC4
;
; Este procedimiento es global.
;*****************************************************************
proc    proc4
        ...
        ret
endp    proc4

;****** CÓDIGO DE TERMINACIÓN ********************************

        end     inicio
```

Macros

Sistemas Digitales con Microprocesadores.

M.C. Juan Carlos Olguín R.

Macros

Una **macro** es un conjunto de instrucciones asociadas a un identificador: el nombre de la macro. Este conjunto de instrucciones es invocado como una sola instrucción o **macroinstrucción.**

Normalmente las instrucciones de una macro se repiten varias veces en un programa o aparecen frecuentemente en los programas. Para emplear una macro en un programa debemos de hacer dos cosas: Definir la macro e invocar la macro.

La **definición de una macro** establece el nombre al que se asocia la macro, el número y nombre de sus parámetros formales y qué instrucciones contiene la macroinstrucción.

La sintaxis de la definición de una macro es la siguiente:

Definición de una macro

```
macro nomMacro [parForm[, parForm]...]
        proposición
       [proposición]
       ...
endm  [nomMacro]
```

donde las directivas **macro** y **endm** marcan el inicio y el final de la definición de la macro. No generan código.

- *nomMacro es el nombre de la macro y se emplea al invocar la macro.*
- *parForm es cada uno de los parámetros formales de la macro. Los parámetros permiten que una* misma macro opere sobre datos distintos.
- *proposición son cada una de las instrucciones que forman la macroinstrucción.*

Aunque la definición de una macro puede ir en cualquier parte de un programa, el lugar más recomendable para su localización es al principio de un archivo, antes de los segmentos de datos y de código.

Al encontrar una **invocación de una macro** el macroensamblador, por ejemplo, substituye la línea con la invocación por las proposiciones que contiene la definición de la macro.

Este proceso de substitución se conoce **expansión de la macro.** La sintaxis de la invocación de la macro es similar a cualquier instrucción:

```
nomMacro [parReal[, parReal]...]
```

donde cada *parReal* es conocido como un ***parámetro real*** de la macro.
Al expandirse la macro cada una de las ocurrencias de un parámetro formal en la definición de la macro se substituye por su correspondiente parámetro real.

Los parámetros reales pueden ser símbolos, mnemónicos, directivas, palabras reservadas, expresiones y números.

Ejemplo sobre macros

El siguiente programa ejecuta todas las rotaciones de un bit con una variable de tipo palabra doble.

Las operaciones de rotación se implementan mediante macros. El parámetro formal **dest** de cada macro representa la localidad de memoria que contiene el dato a rotar.

Al invocar a cada una de las macros el parámetro real de cada macro es el nombre de la variable cuyo dato se va a rotar.

```
;*************************************************************
; ROTACIO2.ASM

; Este programa ejecuta todas las rotaciones de un bit con
; una variable de tipo palabra doble. Las operaciones de
; rotación se implementan mediante macros que rotan el
; contenido de la localidad de memoria dada por el
; parámetro de la macro un bit.
;*************************************************************

;****** CÓDIGO DE INICIO **********************************

        ideal
        dosseg
        model   small
        stack   256
```

```
;****** MACROS ***************************************

;*****************************************************
; RCLDW1
;
; Esta macro rota el contenido de dest un bit a la
; izquierda a través de la bandera de acarreo.
;*****************************************************
macro   rcldw1  dest
        rcl     [word dest], 1          ;; rcl dest, 1
        rcl     [word dest+2], 1
endm    rcldw1

;*****************************************************
; RCRDW1
;
; Esta macro rota el contenido de dest un bit a la
; derecha a través de la bandera de acarreo.
;*****************************************************
macro   rcrdw1  dest
        rcr     [word dest+2], 1        ;; rcr dest, 1
        rcr     [word dest], 1
endm    rcrdw1
```

```
;*************************************************************
; ROLDW1
;
; Esta macro rota el contenido de dest un bit a la izquierda
;*************************************************************
macro   roldw1  dest
        push    bx                          ;; Preserva BX
        mov     bx, [word dest+2]           ;; rol dest, 1
        shl     bx, 1
        rcl     [word dest], 1
        rcl     [word dest+2], 1
        pop     bx                          ;; Restaura BX
endm    roldw1
;*************************************************************
; RORDW1
;
; Esta macro rota el contenido de dest un bit a la derecha.
;*************************************************************
macro   rordw1  dest
        push    bx                          ;; Preserva BX
        mov     bx, [word dest]             ;; ror dest, 1
        shr     bx, 1
        rcr     [word dest+2], 1
        rcr     [word dest], 1
        pop     bx                          ;; Restaura BX
endm    rordw1
```

```
;*** VARIABLES DEL PROGRAMA ********************************

        dataseg
codsal  db      0
r_rcl   dd      ?
r_rcr   dd      ?
r_rol   dd      ?
r_ror   dd      ?

;****** CÓDIGO DEL PROGRAMA ********************************

        codeseg
inicio:
        mov     ax, @data               ; Inicializa el
        mov     ds, ax                  ; segmento de datos

; Rotación a la izquierda a través de la bandera de acarreo.

        rcldw1  r_rcl                   ; rcl r_rcl, 1

; Rotación a la derecha a través de la bandera de acarreo
        rcrdw1  r_rcr                   ; rcr r_rcr, 1

; Rotación a la izquierda
        roldw1  r_rol                   ; rol r_rol, 1

; Rotación a la derecha
        rordw1  r_ror                   ; ror r_ror, 1

salir:
        mov     ah, 04Ch
        mov     al, [codsal]
        int     21h
;****** CÓDIGO DE TERMINACIÓN ********************************
        end     inicio
```

En el ejemplo anterior podemos notar que los comentarios dentro de la definición de una macro empiezan con doble punto y coma. Esto se hace para que en el archivo con el listado del programa generado por el ensamblador no aparezcan los comentarios cada vez que se expande la macro.

También podemos notar que las macros **roldw1** y **rordw1** que implementan las rotaciones a la izquierda y a la derecha, respectivamente, utilizan el registro **BX** y por lo tanto se preserva su valor al inicio de la macro y se restaura al salir.

Esto se hace con el fin de que el uso de la macro sea “transparente” para el usuario.